

Labyrinth Level 1-100

Dokumentation

Ort:	Zürich IBZ
Datum:	05. Oktober 2025
Autor:	Larentis Leandro Andrea
Klasse:	Elektrotechniker HF Semester 4

Inhaltsverzeichnis

1. Managment Summary	2
2. Anforderung / Aufgabestellung	3
2.1. Ziele und Funktionsumfang	3
2.2. Qualitäts- und Nicht-Funktionale Anforderungen	3
2.3. Erwartete Resultate	3
3. Design.....	4
3.1. Spielprinzip und Levelaufbau.....	4
3.2. Systemarchitektur.....	4
3.3. Eigene Ideen und Systemaufbau.....	5
4. Implementierung	6
4.1. Entwicklungsumgebung.....	6
4.2. Struktur des Quellcodes	6
4.3. Algorithmische Umsetzung	6
4.4. Besonderheiten und Optimierungen	7
4.5. Kampagnenmodus	7
5. Test und Resultate	8
5.1. Testumgebung:	8
5.2. Smoke-Test (automatisiert)	8
5.3. Manuelle Tests (Kampagne).....	9
5.4. Analyse der Testergebnisse	10
5.5. Fazit zum Test.....	10
6. Lessons Learned.....	11
6.1. Technische Erkenntnisse.....	11
6.2. Methodische Erkenntnisse.....	12
6.3. Persönliche Erkentnisse	12
7. Anhang	13
8. Umgebung	14

1. Managment Summary

Das Projekt Labyrinth Level 1–100 ist eine praxisorientierte Programmierarbeit, welche die gelernten Konzepte der Programmiertechnik A und B in einem realistischen Anwendungsszenario umsetzt. Entwickelt wurde ein textbasiertes Konsolenspiel in C, bei dem der Spieler (P) in einem ASCII-Labyrinth den Schatz (T) finden muss.

Das Spielfeld wächst mit jedem Level; Hindernisdichte und Mindestabstand zwischen Spieler und Schatz erhöhen sich linear. Über einen BFS-Algorithmus wird jedes Labyrinth automatisch auf Lösbarkeit geprüft.

Das Programm besteht aus mehreren klar strukturierten Modulen (Eingabe, Maze-Generierung, Spielsteuerung, Levelverwaltung) und erfüllt damit die geforderte Modularität und Lesbarkeit. Die Implementierung demonstriert den gezielten Einsatz von Datenstrukturen, Algorithmen und dynamischer Speicherverwaltung.

Die Arbeit zeigt, wie theoretische Grundlagen der Programmierung in eine vollständige Anwendung überführt werden können, und dokumentiert Design, Umsetzung, Tests sowie eigene Ideen zur Level- und Spielfeldentwicklung.

2. Anforderung / Aufgabestellung

Das Projekt *Labyrinth Level 1–100* verfolgt das Ziel, ein vollständiges, in C programmiertes Konsolenspiel zu entwickeln, das die in den Modulen Programmiertechnik A und B erlernten Konzepte praktisch umsetzt.

Das Spiel soll eine dynamische Level-Struktur besitzen und in einer textbasierten Oberfläche ausführbar sein.

2.1. Ziele und Funktionsumfang

- Der Spieler (P) bewegt sich mit den Tasten **W / A / S / D** durch ein ASCII-Labyrinth, um den Schatz (T) zu erreichen.
- Nach jedem Erfolg erhöht sich der Level automatisch, bis Level 100 erreicht ist.
- Die Spielfeldgröße wächst mit dem Level:
 - Level 1–10: 10×10
 - Level 11–25: N×N
 - Ab Level 26: Breite = N, Höhe = 25.
- Die Hindernisdichte steigt linear von 0.12 auf 0.45, bleibt jedoch unter 0.6 für Spielbarkeit.
- Der Mindestabstand (Chebyshev-Abstand) zwischen Spieler und Schatz vergrößert sich pro Level ab Level 10.
- Ein **BFS-Algorithmus** garantiert, dass jedes Level lösbar ist.

2.2. Qualitäts- und Nicht-Funktionale Anforderungen

- Klare modulare Struktur (Trennung von I/O, Maze-Logik, Game-Loop, Levelberechnung).
- Lesbarer, wartbarer und kommentierter Quellcode.
- Portabilität: lauffähig auf ANSI-kompatiblen Terminals (Linux / Windows).
- Farbige Darstellung per ANSI-Codes (abschaltbar über config.h).
- Einfache Kompilierung und Ausführung über Makefile.

2.3. Erwartete Resultate

- Ein funktionsfähiges, spielbares Programm mit 100 aufeinanderfolgenden Leveln.
- Demonstration von Algorithmik (BFS), dynamischer Speicherverwaltung und modularer Architektur.
- Nachvollziehbare Dokumentation von Design, Umsetzung und Tests.

3. Design

3.1. Spielprinzip und Levelaufbau

Das Spiel Labyrinth Level 1–100 ist als textbasiertes Konsolenspiel konzipiert. Der Spieler (P) steuert seine Figur durch ein Labyrinth, um den Schatz (T) zu finden. Nach jedem erfolgreichen Durchlauf wird automatisch das nächste Level gestartet. Die Levelgröße und der Schwierigkeitsgrad steigen schrittweise an.

Levelstruktur:

- **Level 1–10:** Spielfeld 10×10
- **Level 11–25:** quadratische Felder $N \times N$ (z. B. 25×25)
- **Ab Level 26:** Breite = N , Höhe = 25 (z. B. 50×25 bis 100×25)
- **Hindernisdichte:** linear steigend von 0.12 auf 0.45 (max. 0.6)
- **Mindestabstand P–T:** Chebyshev-Abstand, wächst pro Level

Zur Sicherstellung der Spielbarkeit wird nach der Erzeugung jedes Labyrinths mit einem BFS-Algorithmus (Breadth First Search) geprüft, ob ein Pfad vom Spieler zum Schatz existiert. Nur lösbare Labyrinth werden akzeptiert.

3.2. Systemarchitektur

Das Programm folgt einem modularen Aufbau, der die Wartbarkeit und Wiederverwendbarkeit des Codes sicherstellt. Die Module sind über Header-Dateien miteinander verbunden und übernehmen klar abgegrenzte Aufgaben.

Modul und Aufgabe:

config.h	Globale Definitionen (Zeichen, Farben, Standardwerte)
io.c / io.h	Ein-/Ausgabe, Bildschirmsteuerung, Zufallszahlengenerator
level.c / level.h	Berechnung von Spielfeldgröße, Dichte, Mindestabstand je Level
maze.c / maze.h	Erzeugung und Darstellung des Labyrinths, BFS-Prüfung, Bewegung
game.c / game.h	Spiellogik, Level-Wechsel, Anzeige von Level & Zügen
main.c	Programmstart, Parameterübergabe
tests/smoke.c	Unabhängiger Test der Maze-Erzeugung

Die Kommunikation zwischen den Modulen erfolgt über strukturierte Datentypen (Maze, Game, Level) und klar definierte Schnittstellen. Dadurch bleibt die Implementierung übersichtlich und erweiterbar.

3.3. Eigene Ideen und Systemaufbau

Dieses Kapitel beschreibt die individuellen Entwurfsentscheidungen, welche über die Grundanforderungen hinausgehen.

Level- und Spielfeldkonzept

Um den Schwierigkeitsgrad natürlich zu steigern, wächst das Spielfeld anfangs in beide Richtungen ($N \times N$).

Ab Level 26 bleibt die Höhe fix (25 Zeilen), während nur die Breite zunimmt.

So wird das Spiel länger, aber visuell stabil und auf allen Terminals gut lesbar.

Hindernisdichte

Die Waddichte steigt pro Level linear von 0.12 bis 0.45.

Damit vergrößert sich der Anteil der blockierten Felder gleichmäßig, ohne unlösbare Situationen zu erzeugen.

Der BFS-Algorithmus sorgt dafür, dass jedes Level trotz erhöhter Dichte lösbar bleibt.

Mindestabstand (Chebyshev)

Der Abstand zwischen Spieler (P) und Schatz (T) erhöht sich mit dem Level, um ein Fortschrittsgefühl zu erzeugen.

Ab einem gewissen Punkt wird dieser Wert an die Spielfeldhöhe angepasst, damit immer gültige Startpositionen existieren.

Kampagnenfluss

Nach jedem Sieg wird automatisch das nächste Level geladen, wodurch ein kontinuierlicher Spielfluss entsteht.

Das Spiel endet nach Level 100 mit einer Erfolgsmeldung.

Dadurch entsteht eine vollständige Kampagne ohne manuelle Eingriffe des Spielers.

4. Implementierung

Die Umsetzung erfolgte vollständig in C++ (C++17) in GitHub Codespaces mit VS Code. Das Projekt ist plattformunabhängig konzipiert; es benötigt nur einen C++17-Compiler und ein ANSI-fähiges Terminal.

4.1. Entwicklungsumgebung

Programmiersprache	C++
Compiler	g++ 11+/13+ (GNU) oder clang++
IDE	Visual Studio Code (GitHub Codespaces)
OS	Ubuntu Linux (Container in Codespaces)
Build-System	Makefile
Paketierung	Git + GitHub

Codespaces: Devcontainer mit vorinstalliertem g++, cmake, make. Debug/Run direkt im Browser (VS Code) möglich.

4.2. Struktur des Quellcodes

Das Programm ist in mehrere eigenständige Module unterteilt, die über Header-Dateien definiert sind.

config.h	Enthält alle Konstanten, Zeichen-Definitionen und ANSI-Farbcodes
io.c / io.h	Ein-/Ausgabe, Bildschirmsteuerung, Zufallszahlengenerator
level.c / level.h	Berechnung der Spielfeldgröße, Hindernisdichte und Mindestabstände je Level
maze.c / maze.h	Erstellung und Darstellung des Labyrinths, BFS-Prüfung, Bewegung des Spielers
game.c / game.h	Hauptspiellogik, Level-Fortschritt, Zähler, Anzeige
main.c	Einstiegspunkt des Programms, Parametersteuerung
tests/smoke.c	Selbsttest zur Kontrolle der Maze-Funktionalität

Alle Module kommunizieren über klar definierte Schnittstellen und verwenden gemeinsame Datentypen

4.3. Algorithmische Umsetzung

Maze-Erzeugung:

1. Rastergröße und Dichte werden anhand des Levels bestimmt.
2. Das Spielfeld wird zufällig mit Wänden und freien Zellen gefüllt.
3. Der Spieler (P) und der Schatz (T) werden so platziert, dass der Mindestabstand eingehalten wird.
4. Ein BFS-Algorithmus überprüft, ob ein Pfad $P \rightarrow T$ existiert.
5. Wenn kein Pfad gefunden wird, erfolgt eine Neugenerierung, bis ein lösbares Labyrinth entsteht

BFS-Algorithmus (Breadth First Search):

- Durchsucht das Labyrinth schrittweise in vier Richtungen (oben, unten, links, rechts).
- Verwendet eine Queue-Struktur und eine Besuchsmarkierung, um Endlosschleifen zu verhindern.
- Beendet die Suche sofort, sobald der Schatz erreicht ist.

Spielsteuerung:

- Eingaben werden mit `io_read_line()` gelesen und interpretiert.
- Bewegungen werden von `maze_try_move_player()` geprüft (keine Bewegung durch Wände).
- Nach dem Erreichen des Schatzes wird automatisch das nächste Level geladen.

Farbausgabe:

- ANSI-Farben aus `config.h`:
P = rot T = gelb O = grau
- Optional abschaltbar über die Konstante `USE_COLOR`.

4.4. Besonderheiten und Optimierungen

- Verwendung dynamischer Speicherallokation für flexible Spielfeldgrößen.
- Automatisierte Abhängigkeitsverwaltung über Makefile (`-MMD -MP`).
- Konservatives Clamping der Parameter (`density`, `min_sep`) zur Stabilitätssteigerung.
- Einfache Portierung auf Windows-Terminals durch ANSI-Kompatibilität.

4.5. Kampagnenmodus

Nach Abschluss eines Levels erfolgt automatisch der Übergang zum nächsten Level. Das Spiel zeigt in der Kopfzeile den aktuellen Level sowie die Spielfeldgröße an. Erreicht der Spieler Level 100, wird eine Abschlussmeldung ausgegeben und das Programm beendet sich.

Damit entsteht eine in sich geschlossene Kampagne ohne zusätzliche Benutzereingaben.

5. Test und Resultate

Zur Überprüfung der korrekten Funktionsweise wurden sowohl automatisierte Tests als auch manuelle Spieltests durchgeführt.

Die Tests erfolgten direkt im GitHub Codespace-Terminal und konzentrierten sich auf die wichtigsten Funktionsbereiche: Maze-Erzeugung, BFS-Prüfung, Bewegungslogik und Level-Fortschritt.

5.1. Testumgebung:

Entwicklungsumgebung	GitHub Codespaces
Compiler	g++ 11+/13+ (GNU) oder clang++
Terminal	ANSI-fähige Shell (Codespaces VS Code Integrated Terminal)
Build-System	Makefile
Testmethode	Kombination aus Smoke-Test und manuellen Läufen

Die Tests wurden direkt in der Entwicklungsumgebung ausgeführt. Alle Ausgaben erfolgten über das Terminal, da das Spiel keine grafische Oberfläche besitzt.

5.2. Smoke-Test (automatisiert)

Zur Verifizierung der Labyrinth-Erzeugung wurde eine separate Testanwendung smoke.cpp implementiert.

Sie prüft die wichtigsten logischen Komponenten unabhängig vom Spielloop.

Testziel:

- Überprüfung, ob das Labyrinth ordnungsgemäß erstellt und gezeichnet wird.
- Sicherstellung, dass Spieler (P) und Schatz (T) gemäß Mindestabstand platziert werden.
- Kontrolle, ob der BFS-Algorithmus mindestens einen gültigen Pfad findet.

Erwartetes Verhalten:

- Das Programm gibt eine ASCII-Karte aus.
- Am Ende erscheint die Meldung:
Smoke-Test abgeschlossen – Maze wurde erfolgreich erstellt und gezeichnet.

```
@reandos →/workspaces/codespaces-blank/Labyrinth Level 1-100 (main) $ make smoke
gcc tests/smoke.o src/maze.o src/io.o src/level.o -o smoke
🔥 Smoke-Test erstellt. Starte mit: ./smoke
@reandos →/workspaces/codespaces-blank/Labyrinth Level 1-100 (main) $ ./smoke
=== SMOKE TEST: Maze-Generierung ===

Erzeuge Level 15 (15x15, Dichte 0.17, Mindestabstand 10)
000000000000000
0  00  0 0
00 0  0
0 0  0 000 T0
0      000
0      0 0
0 0  0 0
00 00  0
0  0 0 00
0      00
0      0
0      0 0
0 00  0 0
0 P  0 0
000000000000000

✅ Smoke-Test abgeschlossen – Maze wurde erfolgreich erstellt und gezeichnet.
@reandos →/workspaces/codespaces-blank/Labyrinth Level 1-100 (main) $
```


5.4. Analyse der Testergebnisse

Testbereich	Ergebnis	Bewertung
Maze Erzeugung	Fehlerfrei, lösbare Labyrinth	<i>sehr gut (3/3)</i>
BFS-Algorithmus	Korrekte Pfadfindung in allen Fällen	<i>sehr gut (3/3)</i>
Bewegung und Eingabe	Keine Kollisionen / Fehlsteuerung	<i>gut (2/3)</i>
Farbausgabe	Lesbar und stabil in Codespaces	<i>sehr gut (3/3)</i>
Levelsystem	Automatischer Fortschritt Funktioniert	<i>sehr gut (3/3)</i>

5.5. Fazit zum Test

Die Tests haben gezeigt, dass das Spielsystem robust arbeitet und keine kritischen Fehler mehr auftreten.

Der Smoke-Test garantiert die Lösbarkeit jedes Levels, während manuelle Tests den Spielspaß und die Schwierigkeitsskala bestätigen.

Damit ist die Funktionsfähigkeit des Projekts nachweislich gegeben.

Warum aus meiner Sicht bei Bewegung nur gut, weil das mit w, a, s, d Steuerung flüssiger laufen könnte ohne immer a Enter, s Enter usw.

6. Lessons Learned

Im Rahmen der Entwicklung des Projekts *Labyrinth Level 1–100* konnte ich sowohl technische als auch methodische Kompetenzen deutlich erweitern.

Die Arbeit hat gezeigt, wie sich theoretische Programmierkonzepte in eine vollständige, lauffähige Anwendung umsetzen lassen.

6.1. Technische Erkenntnisse

1. Nutzung von GitHub Codespaces

Die Arbeit mit Codespaces war effizient, da eine vollständig vorkonfigurierte Entwicklungsumgebung bereitstand.

Der direkte Zugriff über den Browser, automatische Builds und Terminalintegration haben das Testen und Debuggen erheblich erleichtert.

Ich habe gelernt, Versionskontrolle mit Git aktiv zu nutzen und meine Änderungen sauber zu dokumentieren.

2. Algorithmisches Verständnis

Die Implementierung des Breadth-First Search (BFS)-Algorithmus hat mein Verständnis für Pfadfindung und Laufzeitverhalten erweitert.

Ich konnte erkennen, wie sich komplexe Algorithmen praktisch in Spielmechaniken umsetzen lassen.

3. ANSI-Farben und Konsolensteuerung

Die Verwendung von ANSI-Sequenzen war zunächst ungewohnt, ermöglichte aber eine klare visuelle Strukturierung.

Ich habe gelernt, wie man plattformübergreifende Konsolenausgaben implementiert und optional deaktivierbare Farben integriert.

6.2. Methodische Erkenntnisse

1. Modulare Softwareentwicklung

Ich habe verstanden, wie wichtig eine konsequente Trennung von Logik, Datenhaltung und Darstellung ist. Durch den klaren Modulaufbau (io, maze, level, game) wurde der Code übersichtlicher und leichter wartbar.

Die Arbeit mit Header-Dateien und getrennten Implementationen half, Abhängigkeiten sauber zu managen.

2. Systematisches Testen

Das Arbeiten mit einem **Smoke-Test** und zusätzlichen manuellen Tests hat mir gezeigt, wie Tests zur Qualitätssicherung beitragen.

Ich habe gelernt, Tests nicht als Zusatz, sondern als festen Bestandteil des Entwicklungsprozesses zu sehen.

3. Dokumentation

Die Erstellung dieser Praxisarbeit hat mir verdeutlicht, wie wichtig eine strukturierte technische Dokumentation ist.

Eine klare Trennung zwischen technischen Details (Design, Implementierung) und persönlichen Ideen (Levelsystem) erleichtert das Verständnis für Dritte.

6.3. Persönliche Erkenntnisse

1. Zeit- und Projektmanagement

Ich habe gelernt, meine Arbeit besser zu planen und in Etappen zu strukturieren. Die Aufteilung in Design, Implementierung und Testphasen hat mir geholfen, die Übersicht zu behalten und Deadlines einzuhalten.

2. Eigenständiges Problemlösen

Viele Probleme – etwa bei der BFS-Logik oder der Terminalausgabe – konnte ich selbstständig durch Recherche und Debugging lösen.

Das hat mein Selbstvertrauen in die Softwareentwicklung deutlich gestärkt.

3. Motivation durch sichtbare Ergebnisse

Ein großer Lernfaktor war der Spaß an der Entwicklung.

Das sofort sichtbare Resultat im Terminal motivierte mich, das Spiel immer weiter zu verbessern.

7. Anhang

✓	Labirinth Level 1-100	●
✓	assets	●
≡	sample_maps.txt	U
✓	docs	●
↓	design.md	U
📄	report.pdf	U
✓	include	●
C	config.h	U
C	game.h	U
C	io.h	U
C	level.h	U
C	maze.h	U
✓	src	●
C	game.c	U
C	io.c	U
C	level.c	U
C	main.c	U
C	maze.c	U
✓	tests	●
C	smoke.c	U
M	Makefile	U
ⓘ	README.md	U

8. Umgebung

Ort:

Datum:

Unterschrift(en)

Kerns

05. Oktober 2025

